Tailoring the Scrum Development Process to Address Agile Product Line Engineering

Jessica Díaz, Jennifer Pérez, Agustín Yagüe and Juan Garbajosa

Technical University of Madrid (UPM) - Universidad Politécnica de Madrid
Systems & Software Technology Group (SYST), E.U. Informática, Madrid, Spain
yesica.diaz-at- upm.es, {jenifer.perez,ayague,jgs} -at- eui.upm.es

Abstract. Software Product Line Engineering (SPLE) is becoming widely used due to the improvement it means when developing software products of the same family. However, SPLE demands long-term investment on a product-line platform that might not be profitable due to rapid changing business settings. Since Agile Software Development (ASD) approaches are being successfully applied in volatile markets, several companies have suggested the idea of integrating SPLE and ASD when a family product has to be developed. *Agile Product Line Engineering (APLE)* advocates the integration of SPLE and ASD to address their lacks when they are individually applied to software development. A previous literature review of experiences and practices on APLE revealed important challenges about how to fully put APLE into practice. Our contribution address several of these challenges by tailoring the agile method Scrum by means of three concepts that we have defined: plastic partial components, working PL-architectures, and reflective reuse.

Key words: Agile Product Line Engineering, SPLE, ASD, Scrum, Plastic Partial Components, Reflective reuse

1 Introduction

Many large organizations develop products that can be classified into families. The products of a family share a set of common features and have variable features that make them different. Software Product Line Engineering (SPLE) exploits the commonality across the products of a same family by investing in the upfront design of the product-line platform —i.e. design of the common set of reusable core-assets, their variability and the PL-architecture— (*domain engineering*); then these assets are assembled into customer-specific products just by deriving the existing variability (*application engineering*) [1]. However, this strict *domain-then-application* model, although has been considered successful [1], is resource intensive and risky [2]. Since the product-line platform is planned with a long-term perspective, there is a risk of developing product-line platform that will become obsolete and not used in the products derivation. This risk increases with higher volatility of business situations. Then, not all the efforts in the definition of the product-line platform could be capitalized [3]. The rapid

changing business settings increase frustration to the heavyweight plans, specifications, and other documentation imposed by traditional software development. The deficiencies in practices highlighted above are aggravated when the complexity of product lines grows. Changing market conditions and unplanned changes require alternatives to supplement SPLE. This interest has been expressed by several companies such as Nokia [4] and [5].

Agile Software Development (ASD) may be an alternative to supplement SPLE in changing market conditions, as agile processes harness change for the customer's competitive advantage [6], even late in the development. ASD principles [7] call for incremental development, continuous delivery of valuable software, rapid iterations, and welcome to changes. However, scalability is still a challenge in agile projects; hence, reusability and variability management among the products from the same family are not easily scaled up [8]. As a consequence, an approach resulting from their integration of SPLE and ASD, complementing each other, might make sense. A new approach called *Agile Product Line Engineering (APLE)* [9] advocates their integration with the aim of addressing their lacks when they are individually applied to software development.

Several APLE practitioners [10, 2] have proposed an approach in which the *domain-then-application* model is incrementally iterated. This provides a reactive approach in which reusable core-assets respond to current customer demands while (i) being able to incrementally construct a flexible product-line platform and (ii) being able to deliver features to the customer on time. However, a literature review on the state of the art APLE [11] reveals that the applicability of agile methods to domain engineering requires more effort to meet the challenge of reducing the upfront design with the aim of getting closer to agile principles and values, while achieving the typical goals of SPLE such as reuse [9]. Most approaches converge towards this idea of iteratively and progressively building the product-line platform [12, 13, 5, 2], but in practical terms, there are no mechanisms to incrementally evolve the product-line platform. The agile design of PL-architectures has been pointed out as a key challenge to overcome, since none of the mentioned authors have completely solved it. Dealing with this challenge is essential in order to APLE can be widely adopted by software industry [14]. It seems clear that mechanisms to realize a flexible product-line platform and the strategies to address these mechanisms in the APLE development process, are required.

Our proposal presents a tailored process to address APLE; specifically we focus on the agile method *Scrum*, as it adapts well to software evolution and is widely used by agile community. It is supported through three concepts, some of them, we have previously defined: *plastic partial components* (PPCs) [15] and *working architectures* [16] to iteratively and flexibly design PL-architectures, and *reflective reuse*, that is a new concept to introduce the suitable reuse strategy and reduce the upfront design during the APLE development process.

The remainder of the paper is structured as follows. Section 2 provides an overview of Scrum and APLE. Section 3 defines the concepts and strategies to

tailor the Scrum method to address APLE. Section 4 illustrates the use of our approach. Finally, some conclusions and future work are presented in section 5.

2 Background

2.1 Scrum

Scrum [17] implements an iterative, incremental life cycle which involves three stakeholders: the *Product Owner*, the *Team*, and the *Scrum Master*; all together make up the *Scrum Team*. The Scrum life cycle defines a pre-game phase at the project beginning and iterative planning, review, and retrospective meetings. The *pregame phase* is a light planning process where representative customers and members of the Scrum Team capture requirements as User Stories (US). The US objective is to reduce the cost of the requirement elicitation and management by means of scenarios written by customers without techno-syntax versus conventional methodologies based on formal requirements specification documents. The result is the *product backlog*, a list of known US. Then US are prioritized and divided into short time-framed iterations called *sprints*. A sprint is a 2-4 weeks period of development time. Each sprint has a sprint *planning meeting* at the sprint beginning in which the Product Owner and Team plan together about what to be done for the sprint. The result is the *sprint backlog*, a list of US and tasks that must be performed to achieve the *sprint goal*. At the end of each sprint, the sprint *review* and *retrospective* meetings are held to put into practice continuous improvement. In the review meeting, the Product Owner communicates whether the goals were met, and might introduce changes into the USs. In the retrospective meeting, the Team and the Scrum Master discuss what went well and what could be improved for the next sprint.

2.2 Agile Product Line Engineering

APLE is an attempt of taking the benefits that ASD and SPLE offer in such a way that both can interact and work in cooperation with each other. A systematic literature review on the state of the art APLE [11] reports that APLE would be applicable to business situations in which the convenience of going towards a product line has been identified, but at the same time the market situation is not enough stable for different reasons, including technological and business factors. Specifically, the literature review has identified four main advantages of putting APLE into practice. APLE researches and practitioner have concluded, from their experiences and practices, when APLE could be advantageous:

1. If SPL developers do not have enough knowledge to completely perform the DE, APLE may facilitate the elicitation of further knowledge [12].
2. When anticipated changes cannot be predicted and the SPL life-cycle is not known, it would be advantageous to use an incremental approach such as APLE [18].

3. Agile processes may facilitate fast feedback cycles between requirements engineering, development, and field trial in innovative business [19].
4. Trade-offs between SPLE and ASD provide the opportunity to apply the APLE approach to a wider variety of projects than those served by only applying ASD or SPL methods [12].

McGregor [10] and Hanssen et al. [5] present theoretical attempts to reconstruct a hybrid method from SPLE and ASD principles. They conclude that although at first sight SPLE and ASD may seem contradictory, they actually complement each other. SPLE and ASD can be tailored under the condition that both should retain their basic principles. It seems feasible to tailor SPLE with ASD to obtain an approach that (i) analyzes the most significant commonalities in a domain, rather than an exhaustive set; and (ii) meets changing customer requirements, rather than just simply customizing core-assets [12].

However, since APLE is an emerging approach, organizations have still to face with several barriers to achieve its adoption. Although both the approaches pursue common goals such as improving customer satisfaction and flexibility, and reducing cost and time-to-market, SPLE and ASD apply different strategies to achieve these goals [18, 12, 5]. SPLE stresses the importance of predicting changes at the beginning of the process, and the need of defining a PL-architecture to support customization; in this sense it is capable of accommodating predicted changes to potential members of the product line. On the other hand, ASD emphasizes value delivery to the customer and welcomes changes by means of incremental development and close iterations with customers. ASD advocates for minimal investment in an upfront architecture when knowledge is not readily available, and encourage the continuous improvement and refactoring of the architecture to achieve the business goals. In fact, agile methods have a reputation for paying very little attention to software architecture [5] and some agile practitioners even advocate against investing effort in architecture specification as it is perceived as wasted effort [6].

Tian et al. [12] explicitly recognized these potential challenges and risks associated with APLE: (i) traceability management and maintenance of components might be difficult in agile approaches without explicit knowledge; and (ii) if PL-architectures are tailored to be more agile, there is a danger that a valuable architecture supporting other products of the family may be damaged. This paper focuses on the last one. In fact, the agile design of PL-architectures has been pointed out as a key challenge to overcome, since none of the mentioned authors have completely solved it. Dealing with this challenge is essential in order to APLE can be widely adopted by software industry [5, 14]. The key question is: *how to take care of the long-term planning and upfront architecture required by SPLE while still being able to deliver value to the customer in time?* [14]; or from the opposite point o view: how architecture can be tailored to be more agile without losing the SPL reusability and flexibility?

3 Tailoring Scrum to address APLE

This section presents how APLE is tackled by means of a tailored Scrum in which DE and AE processes are performed in an iterative way. To overcome the PL-architecture challenge and reducing the upfront design, it is necessary to define the mechanisms and strategies to be applied during the APLE development process. This is why our contribution is based on these three concepts: *plastic partial components* (PPCs), *working PL-architecture*, and *reflective reuse*. The first ones have been previously defined in [15, 16] and provide the mechanism to easily evolve PL-architectures in an agile context, and the last one is a new concept that provides the means to introduce the suitable reuse strategy during the development process. These concepts and the tailored Scrum for APLE are detailed below.

3.1 Plastic Partial Component (PPC) & Working PL-Architecture

PL-architectures define the common and variable structure for all products of a SPL. This means that PL-architectures can rapidly respond to variabilities defined by different stakeholders and changes within a well defined market segment. The APLE development process requires that PL-architectures (i) are incrementally and iteratively designed, and (ii) welcome unplanned changes. This means that the PL-architecture has to evolve in each of the iterations to incrementally include all the features of the product-line. From these assumptions, we realized that it is necessary to be flexible enough for supporting not only *external evolution* (changes in the architectural configuration by adding or removing components or connections), but also *internal evolution* (changes inside components).

Our contribution is based on a previous work [16] in which we attempted to solve the problem of designing software architectures in ASD (*agile architecting*) based on the PPC concept. PPCs address agile architecting by defining architectural components in a iterative and incremental way. PPCs are highly malleable components that can be partially described, which increases the flexibility of architecture design. In fact, the notion of PPC was originally defined for SPLE to support the definition of internal variation of architectural components [15], i.e. specification of variants by modifying the configuration of the software architectures (adding/removing components to/from the architecture), but also specification of variations inside components. PPC variability mechanism is used to define variations among products, and to flexibly add, remove, and modify the *working PL-architecture* throughout the APLE life cycle. A working PL-architecture is one that is incrementally developed with the working products of the SPL, and includes only the features which are necessary for the current iteration. As a result, working PL-architectures allow keeping the system in sync with changing conditions.

The variability mechanism underlying PPCs is based on the principles of *invasive software composition* (ISC) [20]. ISC defines components as fragment boxes that hook a set of reusable fragments of code. Specifically, ISC proposes

that these fragments of reusable code can be aspects which make components, and by extension, software architectures, easier to maintain. Following these principles, we defined the variability of a PPC using *variability points*, which hook fragments of code to the PPC known as *variants*. These variants, as well as components and PPCs, realize the requirements —in terms of features— that have been defined by the domain analysis process at the architectural level. As requirements can be related to concerns that crosscut the software architecture (crosscutting concerns) or not (non-crosscutting concerns), we have stated that those variants that realize crosscutting concerns are called *aspects*, and those that realize non-crosscutting concerns are called *features*. Variability points allow us to specify the *weaving* between the PPC and the variant. The weaving principles of Aspect-Oriented Programming (AOP)[21] provide the needed functionality to specify *where* and *when* extending the PPCs using variants. Therefore, a PPC is defined by specifying (see Figure 1): (i) its variability points; (ii) the aspects and/or features that are necessary to complete the definition of the component for any product of the family; and (iii) the hooks between the variability points and the aspects and/or features. The complete definition of a PPC for a specific product is created by means of the selection of aspects and/or features through the variability points. A formal and more complete definition of the PPC, its *metamodel* and its *graphical metaphor* can be found in [15].

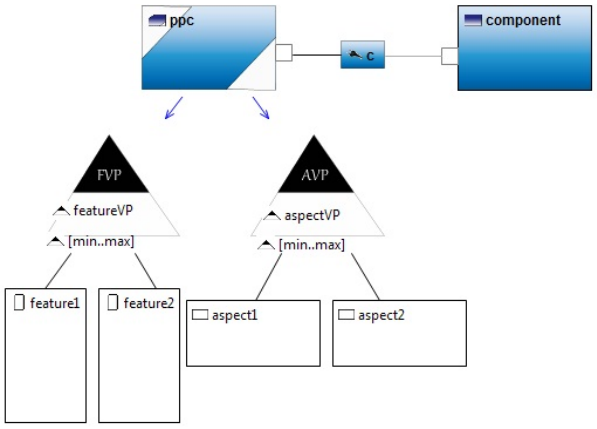

Fig. 1. PPC example

Therefore, PPCs variability mechanisms can be advantageously used for evolving working PL-architectures throughout the different iterations that comprise the APLE development. PPCs behave as extensibility mechanisms to flexibly compose pieces of software (aspects, features, components) as if we were building a puzzle. Their characteristics *partial* and *plastic* are fundamental to support the APLE development. As PPCs are partial, they can be incompletely specified, in such a way that they may be completely specified in further iter-

ations. PPCs allow one to incrementally develop architectural components by only taking into account the required functionality for each iteration. As PPCs are plastic, they are highly malleable. This is thanks to their mechanisms for specifying variability, which allow one to flexibly adapt software component by easily adding, or removing fragments of code. Since the *weaving* principles of AOP provide the needed functionality to specify *where* and *when* extending the PPC through using the variants, variants are independent of the component of which they hang. Therefore PPCs architectural components are ready to be extended or modified at any moment. As a result, PPCs allow getting closer and closer to agile values and principles. Finally, the description of working PL-architectures using PPCs is supported by a graphical modeling tool called Flexible-PLA[1]. Flexible-PLA is an open-source tool that has been developed following the MDD approach using the Eclipse Modeling Framework (EMF) and its Graphical Modeling Framework(GMF).

3.2 Reflective Reuse

Barriers to reuse have been reported thoroughly in the literature [22, 23]. Our contribution focusing on dealing with the initial upfront investments for constructing repositories by means of the *reflective reuse* concept. Reflective reuse takes a step forward providing a theoretical foundation and defining the needed changes in software development processes. But before defining what reflective reuse is, systematic and opportunistic reuse are briefly revisited.

Systematic reuse is a preplanned process to develop software to be reused in a long term, i.e. software is proactively implemented anticipating future customer needs. This software is usually stored in repositories that are available during the development process to be reused. Systematic reuse consists of reusing from these repositories as many times as it is possible following a *reuse-planning*. This kind of reuse requires a high investment due to the construction of repositories. This investment is capitalized once a repository has a reasonable quantity of reusable software, which can be systematically reused in the development and maintenance of companies' products.

Opportunistic reuse is a reactive low-cost process to improve the efficiency of development in the short term. It is based on opportunities for reuse demonstrating better results to rapidly develop innovative software products in small to medium-sized organizations. Opportunistic reuse is a process to extend software with functionality from a third-party software supplier but this functionality was not originally intended to be integrated and reused [24, 25]. Therefore, opportunistic reuse is not based on a preplanned reuse and they do not invest in the construction of repositories. However, opportunistic reuse fails to effectively deal with the search-and-procurement processes of artifacts.

A representative example of the application of systematic reuse is the SPLE approach. Whereas, ASD does not support systematic reuse because it focuses on immediate demands of customers assuming that future changing scenarios

[1] Available on https://syst.eui.upm.es/FPLA/home

cannot be anticipated in any case, and the up-front investment for the definition of a repository will not be profitable. But ASD does not preclude other approaches based on short-term benefits of software reuse such as opportunistic reuse. It would be greatly useful to be able to define an approach in which opportunistic and systematic reuse can be applied without paying the cost of giving up some of the essential advantages of one of them. From reactively opportunistic reuse to proactively systematic reuse there is a range of reuse strategies based on the required anticipating degree. *Reflective reuse* is an abstract concept that encompasses this range from opportunistic to systematic reuse strategies.

Reflective reuse is necessary in APLE to support different reuse needs during the development process. Reuse in APLE is not preplanned in the sense that it is not an activity that entails the whole development process, as in the case of traditional SPLE, otherwise it only plans a short time (an iteration). So, the repository is built based on an analysis of future opportunities for reuse in the short time. Of course a long-term vision is present, but this vision does not compel the whole process. Since the construction of the repository does not require initial upfront investment, risk of no being profitable is reduced. Reflective reuse also takes advantages of opportunities and flexibility generated by changing requirements when systems have to be quickly evolved and adapted.

3.3 APLE Scrum development process

Our vision of APLE involves a tailored Scrum process in which each sprint consists of two sub-processes: a first one for DE and a second one for AE. These sub-processes support the incremental definition of the variability, modeling and design of the working PL-architecture, development of the set of reusable core-assets, and derivation of the working products of the SPL. The tailored Scrum process for addressing APLE is detailed as follows:

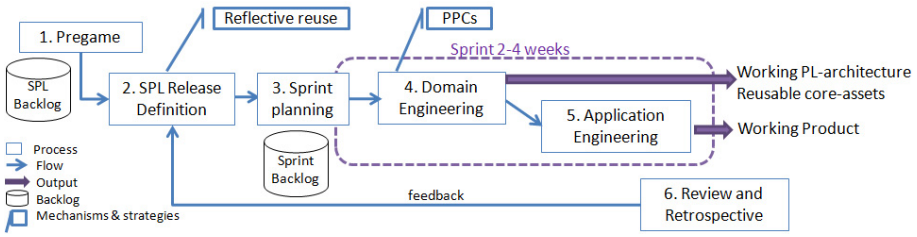

Fig. 2. APLE Scrum development process

In the *pregame* phase (see Figure 2 box 1), representative stakeholders and the Scrum Team capture and analyze requirements of the SPL in terms of USs. The result of this preplanning is a list of known user stories called *SPL backlog*. Then, the SPL backlog is analyzed in terms of features, this is, logical units of behavior that are specified by a set of USs. Common and variable features are

explicitly documented in feature models, which is an abstraction of the variability [26]. This first feature model is not exhaustive and it is consolidated during the next sprints; its objective is to provide the scope of the SPL.

In the *SPL Release Definition* phase (see Figure 2 box 2), features are prioritized and divided into sprints. The goal is optimizing the features to be considered in each sprint. Common features can be prioritized from the opportunity to be reused in the short term following the definition of reflective reuse. Variable features can be prioritized on the basis of the lowest change impact, the highest value to the market or flexibility to react to changing market situations.

In the *Sprint planning* meeting (see Figure 2 box 3), the features to be implemented in the current sprint are planned and estimated. It determines the *sprint goal* and the *sprint backlog*.

In the *DE* phase (see Figure 2 box 4), the feature model, PL-architecture, and core-assets, are incrementally consolidated in each sprint. To achieve it, the concepts PPC defined in Section 3.1 is applied to. The PPCs, variability points, variants, and connectors are defined to construct the *working PL-architecture*.

In the *AE* phase (see Figure 2 box 5), variability points are bound to specific variants. This consists in selecting and applying the variants that are required for a/several specific working product/s —which depends on the sprint goal defined in the sprint planning meeting. It can be divided into two different tasks: (i) to reconfigure the working PL-architecture by adding or removing the variable components, and connectors; and (ii) to complete the partial specification of PPCs by weaving aspects and/or features with the core functionality of the PPC. The results of this phase are: the *working product-architecture/s* and the *working product/s*.

Finally, the *review* and *retrospective* meetings (see Figure 2 box 6) provide feedback to apply the needed changes and adjustments to the next sprint. Changes in USs enforce to re-prioritize the SPL backlog and allow being aware of new opportunities for reuse. As a result, the reuse process is able to be flexible when systems have to be quickly evolved. It lies the concept of reflective reuse.

Therefore, the APLE Scrum process uses the PPCs and working PL-architecture concepts to construct a flexible SPL platform. Finally, reflective reuse provides the foundations to implement the best reuse strategy during the development on the basis of the opportunities of reuse in the short-time or taking advantages of systematic reuse in the long-term.

4 Experience Report

This section illustrates the application of the tailored Scrum process that we have proposed to address APLE. It reports our experience in developing environments for testing software systems (*Test and OPeration ENvironment*, TOPEN [27]). Since testing different *systems under test* (SUTs) requires different features of

TOPEN and since SUTs are constantly evolving, APLE is an appropriate approach to address this software development[2].

The definition of the TOPEN Product-Line has been explained thoroughly in [27]. This case study focuses on a TOPEN product for testing testing biogas power production plants (*TOPEN Biogas*). Following the APLE Scrum process defined in Section 3, TOPEN Biogas was developed through 6 SCRUM sprints during 15 weeks. During the pregame phase, several USs were identified: (US1) *Test engineers specify a test case utilizing a user interface and with the biogas plant specific language.* (US2) *Test engineers compile and execute a test case from the user interface. The results of the test case executions must be shown to them.* (US3) *Test engineers remotely test/monitor the biogas plant.* Next, the SPL backlog was created, USs were analyzed in terms of features, and an initial feature model was performed (see Figure 3). Then, features were prioritized and divided into sprints on the basis of the higher value for the customer.

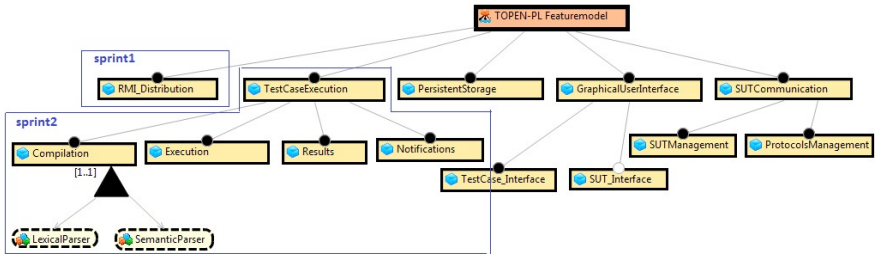

Fig. 3. TOPEN product-line Feature Model

After analyzing these feature, the architects identified four main PPCs that made up the working TOPEN PL-architecture: the *Graphical User Interface* (GUI), the *Topen Engine*, the *Data Management System* (DMS), and the *Gateway*. The GUI offers test engineers the graphical interface to define and execute test cases. The Topen Engine compiles and translates test cases which are defined by test engineer in a domain-specific test language into the language that the SUT is able to process. The DSM stores test data and maintain the system business rules. And, the Gateway establishes the communication between the Topen Engine and the SUT.

This report focuses on the PPC Topen Engine to illustrate how the working TOPEN PL-architecture was iteratively evolved in each sprint. The PPC Topen Engine was implemented during the first sprints because architects were aware of the opportunities of reusing in the short-term to easily adapt TOPEN to other SUTs. In the first sprint, architects focused on the aspect *distribution* that it was firstly based on the *remote method invocation* (RMI) (see Figure 4a). In the second sprint, architects focused on the *Semantic* and *Lexical* parsers (see Figure

[2] This project has been developed with the collaboration of the company Answare Tech within Flexi ITEA2 6022 project, whose goal is to scale agile.

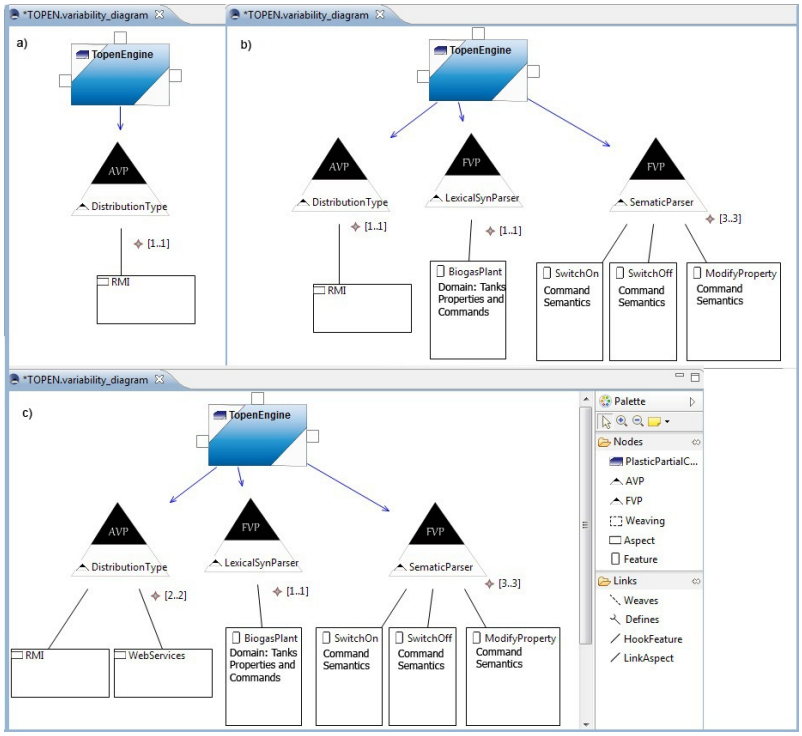

Fig. 4. Working TOPEN PL-architectures

4b). These parsers validate both lexically and semantically the commands and alarms that test engineers send or receive to/from the biogas plant. The working TOPEN PL-architecture was evolved to support these parsers by adding two variability points and their respective variants. Although it implied more effort in specifying the pointcuts and the weaving operators, we gained in code modularity, scalability and reusability because these fragments of code are unaware of the linking context. These variability points may provide flexibility to evolve the working TOPEN PL-architecture to other language in the short-term. In a third sprint, a change in the specifications forced to implement *web services* to support the communication between the four main PPCs. It was solved by hooking a new distribution aspect, which provided web services technology (see Figure 4c). Hence, the distribution of our application was modified just by adding a new single fragment of code. As a result, the incremental design of the working TOPEN PL-architecture by means of the use of PPC allowed us to flexibly modify the distribution capabilities with the minimum impact both in cost and effort, and to inherently refactor the distribution code. Following sprints tackled with all the features of the TOPEN Biogas tacking the advantages of PPC and reflective reuse as we have shown in these first sprints.

5 Related Work

O'leary et al. [28] define an *agile framework for product derivation* (AFPD) which assists organizations in integrating agile practices in the product derivation process. AFPD considers the adoption of early and continuous delivery strategy, automation of product derivation, product derivation iterations, and agile testing techniques. Babar et al. (P36) describe a successful industrial case study, and analyze the organizational processes and practices that were used to integrate SPLs and ASD. The authors describe a development process that consists of three sub-processes: product line platform, exploration before agile product development, and agile product development. Although these two approaches are valid and compelling, they focus on a reuse-centric AE process, and do not discuss the role of agile methods in DE. As a result, they are closer to the concept of agile development using a SPL platform than the concept of APLE —which tries to address the entire process, i.e. DE and AE.

Paige et al. [13] introduce the agile design of PL-architectures. They propose that architecture should be incrementally designed, and variations should be generated as a result of the agile refactoring practice. Even though this proposal shows potential, considerations about incremental design of PL-architecture were not described. Meanwhile, Kakarontzas et al. [29] present an approach based on *elastic components* to specify variability. Elastic components address the configuration and evolution of components by means of adding/deleting/modifying variants that hook from the root component. This work is a first step to show how PL-architectures can be tailored to be more agile. However, since their variants are context-dependent, additional mechanisms are needed to make variants independent of the root component. Context-independent variants may (i) facilitate the incremental and iterative evolution of PL-architectures in ASD by reducing the number of changes; and (ii) make more flexible the reuse of variants among the products of a SPL. In our proposal, variants are unaware of the linking context, and they are completely reusable.

Finally, Ghanam et al. [2, 30] highlight the importance of mining systems, bearing in mind reuse, and the formalization of commonality and variability through acceptance tests. They introduce an iterative model for APLE and the use of test-driven development (TDD) to support this process. Specifically, they propose a bottom-up application-driven approach that relies on automated acceptance tests to derive core-assets from existing code. Through this bottom-up approach, the SPL is iteratively built from existing product instances, the product-line platform progressively evolves, and variability is handled on-demand, i.e. reactively. [30] presents a tool for assisting the refactoring of code when a developer introduces a variation. The code-based tool requests to developers the method that is causing the variation and creates an abstract factory for this method, the corresponding concrete classes, and the required test classes. This approach provides a novel contribution based on TDD to model variability dealing with DE and AE in a context where the XP method is used. Despite the fact that this contribution is a significant advance in the area, more work is still necessary to support coarse-grained variability, i.e. not always it is possible to

trace a variation with a class or a method. A variation could be traced to a set of methods, classes, attributes, or even components, and therefore, a mechanism such as PPCs is required to easily evolve PL-architectures.

6 Conclusions

It can been concluded that APLE as an approach is feasible, although some challenges still remain open. For addressing APLE, within the worked reported, the concepts of PPCs, working PL-architectures, and reflective reuse, have been used to tailor the agile method Scrum. As future work, we plan to use APLE in larger-size projects, specifically in the power smart grid application domain in order to obtain measures from empirical results.

References

1. Pohl, K., Bckle, G., Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Germany (2005)
2. Ghanam, Y., Park, S., Maurer, F.: A test-driven approach to establishing and managing agile product lines. In: SPLiT'08 in conjunction with SPLC'08. (2008)
3. Verlage, M.: The invisible man-month or what is the real value of a core asset? http://splc2007.jaist.ac.jp/keynotes.html (2007)
4. Känsälä, K.: Good-Enough Software Process in Nokia. Product Focused Software Process Improvement **3009/2004** (2004) 424–430
5. Hanssen, G.K., Fígri, T.E.: Process fusion: An industrial case study on agile software product line engineering. J. Syst. Softw. **81**(6) (2008) 843–854
6. Cockburn, A.: Agile Software Development. The Cooperative Game. Second Edition. Addison-Wesley Professional (2006)
7. K. Beck et al.: Manifesto for agile software development
8. Ghanam, Y., Andreychuk, D., Maurer, F.: Reactive variability management in agile software development. In: Agile '10: Proceedings of International Conference on Agile Methods in Software Development, IEEE Computer Society (2010) 27–34
9. Cooper, K., Franch, X.: Aple 1st international workshop on agile product line engineering. In: SPLC '06: Proceedings of the 10th International on Software Product Line Conference, IEEE Computer Society (2006) 205–206
10. McGregor, J.D.: Agile software product lines, deconstructed. Journal of Object Technology **7**(8) (November-December 2008) 7–19
11. Díaz, J., Pérez, J., Alarcón, P.P., Garbajosa, J.: Agile product line engineeringa systematic literature review. Software: Practice and Experience Journal **41**(8) (2011) 921–941
12. Tian, K., Cooper, K.: Agile and software product line methods: Are they so different? In: APLE '06: 1st International Workshop on Agile Product Line Engineering. (2006)
13. Paige, R., Wang, X., Stephenson, Z., Brooke, P.: Towards an agile process for building software product lines. In: XP '06: Proceedings of Extreme Programming and Agile Processes in Software Engineering. Volume 4044/2006., Berlin, Heidelberg, Springer-Verlag (2006) 198–199

14. Ali Babar, M., Ihme, T., Pikkarainen, M.: An industrial case of exploiting product line architectures in agile software development. In: SPLC '09: Proceedings of the 13th International Conference on Software Product Lines. (2009)
15. Pérez, J., Díaz, J., Costa-Soria, C., Garbajosa, J.: Plastic partial components: A solution to support variability in architectural components. In: WICSA'09: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society Press (2009) 221–230
16. Pérez, J., Díaz, J., Garbajosa, J., Alarcón, P.P.: Flexible working architectures: agile architecting using PPCs. In: ECSA'10: Proceedings of the 4th European conference on Software Architecture, Berlin, Heidelberg, Springer-Verlag (2010) 102–117
17. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice-Hall (2002)
18. Carbon, R., Lindvall, M., Muthig, D., Costa, P.: Integrating product line engineering and agile methods: Flexible design up-front vs. incremental design. In: APL'06: 1st International Workshop on Agile Product Line Engineering. (2006)
19. Kircher, M., Schwanninger, C., Groher, I.: Transitioning to a software product family approach - challenges and best practices. In: SPLC '06: Proceedings of the 10th International on Software Product Line Conference, Washington, DC, USA, IEEE Computer Society (2006) 163–171
20. Assmann, U.: Invasive Software Composition. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2003)
21. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (2001) 327–353
22. Favaro, J.M., Favaro, K.R., Favaro, P.F.: Value based software reuse investment. Annals of Software Engineering **5** (1998) 5–52
23. Frakes, W.B., Kang, K.: Software reuse research: Status and future. IEEE Trans. Softw. Eng. **31**(7) (2005) 529–536
24. Kotonya, G., Lock, S., Mariani, J.: Opportunistic reuse: Lessons from scrapheap software development. In: CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering, Springer-Verlag (2008) 302–309
25. Jansen, S., Brinkkemper, S., Hunink, I., Demir, C.: Pragmatic and opportunistic reuse in innovative start-up companies. IEEE Softw. **25**(6) (2008) 42–49
26. K. Kang et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI-90-TR-21, Carnegie Mellon University, PA (1990)
27. Magro, B., Garbajosa, J., Pérez, J.: Development of a Software Product Line for Validation Environments Software. In: Applied Software Product Line Engineering. Taylor and Francis (2009) 173–199
28. O'Leary, P., McCaffery, F., Richardson, I., Thiel, S.: Towards agile product derivation in software product line engineering. In: EuroSPI'09: Proceedings of 16th European Conference on Software Process Improvement. (2009) 8.1–8.6
29. Kakarontzas, G., Stamelos, I., Katsaros, P.: Product line variability with elastic components and test-driven development. In: CIMCA '08: Proceedings of the 2008 International Conference on Computational Intelligence for Modelling Control & Automation, IEEE Computer Society (2008) 146–151
30. Ghanam, Y., Maurer, F.: Extreme product line engineering refactoring for variability: A test-driven approach. In: XP '10: Proceedings of the 11th International Conference Agile Processes in Software Engineering and Extreme Programming. Volume 48 of LNBIP., Springer (2010) 43–57